A TASK CONCURRENCY MANAGEMENT DESIGN METHOD

Field of invention

[0001]   The invention relates to system level hardware-software codesign method especially including the management of concurrent tasks.

Background of the invention

[0002]   Semiconductor processing technology nowadays allows complex systems to be integrated on one single chip. A consistent system design technology is in great need to cope with such complexity and with the ever shortening time-to-market requirements (B. Tuck Raise your sights to the system level - design report: '97 Paris forum (round-table discussion). *Computer Design,* pp. 53-74, June 1997.) . It should allow to map these applications cost-efficiently to the target realisation while meeting all real-time and other constraints.

[0003]   The target applications of our task-level system synthesis approach are advanced real-time multi-media and information processing (RMP) systems, such as consumer multi-media electronics and personal communication systems. These applications involve a combination of complex data- and control-flow where complex data types are manipulated and transferred. Most of these applications are implemented with compact and portable devices, putting stringent constraints on the degree of integration (i.e. chip area) and on their power consumption. Secondly, these systems are extremely *heterogeneous in nature* and combine high performance data processing (e.g. data processing on transmission data input) as well as slow rate control processing (e.g. system control functions), synchronous as well as asynchronous parts, analog versus digital, and so on. Thirdly, time-to-market has become a critical factor in the design phase. Finally, these systems are subjected to stringent real-time constraints (both hard and soft deadlines are present), complicating their implementation considerably.

[0004]   The platform for these applications include one or more programmable components, augmented with some specialized data paths or co-processors (accelerators). The programmable components run software components, being slow to medium speed algorithms, while time-critical parts are executed on dedicated hardware accelerators.

[0005]   When looking at contemporary design practices for mapping software (and hardware) on such a platform, one can only conclude that these systems nowadays are designed in a very *ad hoc* manner (F.Thoen, F.Catthoor, "Modeling, Verification and Exploration of Task-level Concurrency in Real-Time Embedded Systems," ISBN 0-7923-

7737-0, Kluwer Acad. Publ., Boston, 1999.) The design trajectory starts by identifying the global specification entities that functionally belong together, called tasks or processes. This step is followed by a manual 'hardware-software partitioning'. Because of separate implementation of the different tasks and of the software and hardware, afterwards a *system integration* step is inevitable. This manual step performs the 'system/software embedding' and synthesizes the interface hardware, which closes the gap between the software and the hardware component.

[0006]   The main goal of system/software embedding is to encapsulate the concurrent tasks in a control shell which takes care of the *task scheduling (software scheduling in the restricted sense)* and the inter-task communication. Task scheduling is an error-prone process that requires computer assistance to consider the many interactions between constraints. Unfortunately, current design practices for reactive real-time systems are *ad hoc* and not very retargetable. Designers have used real-time operating systems (RTOS) or kernels to solve some of these scheduling problems. Both of RTOSs and kernels assume a specific processor and a particular I/O configuration. Such practices result in poor modularity and limited retargetability, thus severely discourage exploitation of the co-design space. This is the case even if the program is written in a high-level language. Moreover, these RTOSs in fact only provide limited support for real-timeness, and leave satisfaction of the timing constraints to the designer. They can be considered as nothing more than an optimized back-end for performing the task scheduling, and typically they are not integrated in a design framework in which a global specification model serves as entry point.

[0007]   Existing approaches we will work neither at the detailed *white-box* task model (see e.g. P.Eles, K.Kuchcinski, Z.Peng, A.Doboli, P.Pop, "Scheduling of conditional process graphs for the synthesis of embedded systems," *Proc. 1st ACM/IEEE Design and Test in Europe Conf.*, Paris, France, pp.132-138, Feb. 1998. P.Hoang, J.Rabaey, "Scheduling of DSP programs onto multiprocessors for maximum throughput," *IEEE Trans. on Signal Processing*, Vol.SP-41, No.6, pp.2225-2235, June 1993.) where all the operations are considered already during the mapping and where too much information is present to allow a thorough exploration, nor at the *black-box* task model (see e.g. S.Ha, E.Lee, "Compile-time scheduling of dynamic constructs in dataflow program graphs," *IEEE Trans. on Computers*,

3

Vol.C-47, No.7, pp.768-778, July 1997. I.Hong, M. Potkonja, M. Srivastava, "On-line scheduling of hard real-time tasks on variable voltage processor," *Proc. IEEE Int. Conf. on Comp. Aided Design*, Santa Clara CA, pp.653-656, Nov. 1998. Y.Li, W.Wolf, "Scheduling and allocation of single-chip multi-processors for multimedia," *Proc. IEEE Wsh. on Signal Processing Systems (SIPS)*, Leicester, UK, pp.97-106, Nov. 1997. where insufficient information is available to accurately steer even the most crucial cost trade-offs.

[0008] Other work P.Eles, K.Kuchcinski, Z.Peng, A.Doboli, P.Pop, "Scheduling of conditional process graphs for the synthesis of embedded systems," *Proc. 1st ACM/IEEE Design and Test in Europe Conf.*, Paris, France, pp. 132-138, Feb. 1998.) considers task scheduling as a separate issue from cost. In their work, the assignment to processors with different power or varying supply voltage is either "automatic," i.e., without trade-off between timing and cost, e.g., the processor energy costs, or not treated.

Aim of the invention

[0009]   It is the aim of the invention to present an approach for management of concurrent tasks as part of a system level hardware-software codesign being capable of exploring both timing and processor cost issues with a reasonable complexity.]

Summary of the invention

[0010]   It is an aspect of the invention to incorporate in the design steps of an essentially digital system a two-level scheduling approach. In said approach said description describes the functionality and timing aspects of said digital system and comprises of a set of or plurality of communicating tasks while each of said tasks comprise of a set of or a plurality of nodes. With two-level scheduling approach is meant that first for at least one of said tasks its nodes are scheduled. When scheduling of at least one of said tasks is performed, scheduling of tasks itself is performed. In an embodiment of said two-level scheduling approach scheduling within tasks, thus of task nodes can be denoted static scheduling. Said task scheduling can be denoted dynamic scheduling. The terminology static and dynamic finds its origin in the fact that the so-called grey-box is constructed by placing deterministic, thus rather static behavior in nodes while non-deterministic behavior, thus rather dynamic behavior, is placed in tasks. In an embodiment of said two-level scheduling approach said scheduling within tasks, thus of task nodes includes besides time constraints also cost optimization, more in particular said cost is related to the processors on which said task nodes are executed. Said two-level scheduling includes thus besides ordening of said task nodes, being a typical output of scheduling, also assignment of task nodes to processors. In said two-level scheduling a concurrent architecture with a set or a plurality of processors is assumed. Examples are given for two processor architectures below but the invention is not limited thereto. In an embodiment of said two-level scheduling approach said static scheduling results in a plurality of partial static scheduled description. Said partial static scheduled descriptions or partial static schedule options are further exploited in said dynamic scheduling, meaning that said dynamic scheduling performs a step of selecting for said tasks one of said partial static schedules. Further design of said essentially digital system is then performed based on the partial scheduled description resulting from said dynamic scheduling step. Each static scheduling step generates cost-cycle budget trade-off curves which are exploite by said dynamic scheduling step. Alternatively one can state that said static scheduling step generates scheduling options, being different in cost and cycle budget, each option being related to a particular scheduling of nodes and allocation of nodes on processors.

Said dynamic scheduling step selects for each of said tasks one of said options while guaranteeing that said options are compatible, meaning taking into account the limited resources or processors into account. It is an aspect of the invention to incorporate in the design steps of an essentially digital system a step of modifying a description of said essentially digital systems with task concurrency improvement transformations. Said description describes the functionality and timing aspects of said digital system and comprises of a set of or a plurality of communicating tasks while each of said tasks comprise of a set of or a plurality of nodes. The modifying step transforms a first or initial description into a second or improved description. Said transformation are aimed at improving the concurrency between said tasks. With concurrency is meant the ability to execute tasks at least partial in parallel. Often within said first or initial description hiddes that some tasks have concurrent behavior, meaning have the ability to be executed partially simultaneously. So hidding of concurrency is most often due to data dependencies. However often within said first description data dependencies which are not functionally necessary are introduced, thereby preventing concurrency of tasks unnecessary. Said task concurrency improvement transformations comprises of a data dependency removing step while maintaining the functionality of the digital system, possibly by introducing some buffering, meaning introducing some storage space for data. It can be noted that although concurrency improvement of tasks is aimed at, said concurrency improvement steps can have as extra effect that concurrency of nodes within tasks is improved also. In an embodiment said task concurrency improvement transformations are used on said descriptions used in said two-level scheduling approach, more in particular before performing static scheduling. It is an aspect of the invention to recognize that task concurrency management should be performed early in the design flow for designing essentially digital system. With early is meant before hardware-software partitioning, thus before deciding which tasks will be performed by running software on a predefined processor (software implementation) and which tasks will be performed by the running of a dedicated custom, meaning further to be designed, processor. It must be noted that the processor representation used in the schedulings are only abstractions of the later physical realisation of the system. The invention can be formalized as follows: A method for designing an essentially digital system, comprising the steps of:

7

[0011] 1. describing the functionality and timing of said digital system in a description being a set of (at least two) or a plurality of thread frames (also denoted tasks), each of said thread frames being a set of (at least two) or a plurality of thread nodes;

[0012] 2. generating a plurality of static schedulings for at least two of the thread nodes of one of said thread frames;

[0013] 3. performing a dynamic scheduling on at least two of said thread frames, said dynamic scheduling comprising of selecting for at least one of said thread frames one of said generated plurality of static schedulings, said dynamic scheduling resulting in an at least partial scheduled description; and

[0014] 4. designing said digital system from said at least partial scheduled description.

[0015] The method described above, wherein said essentially digital system being modelled as a set of or a plurality of processors and said generating of a plurality of static schedulings comprising solving of a first time-constraint cost optimization problem wherein thread nodes are assigned to said processors and said cost essentially being related to said processors. The method described above, wherein said generating of a plurality of static scheduling generates cost-cycle budget trade-off curves for at least two thread frames, said cost-cycle budget trade-off curves comprise of points representing said static schedulings. The method described above, wherein said dynamic scheduling comprises of a second time-constraint cost optimization problem. The method described above, wherein said first time-constraint cost optimization optimizes a cost function comprising at least energy consumption of a processor executing a scheduled thread node. The method described above, wherein said second time-constraint cost optimization optimizes a cost function comprising at last energy consumption of a processor Executing a scheduled thread node. The method described above, wherein said generating of static schedulings for a thread frame being capable of modifying the amount of thread nodes within said thread frame. The method describing above, wherein said tasks are communicating. The method described above, wherein non-deterministic behavior of said essentially digital system is described by the dynamical scheduling of the thread frames while the deterministic behavior is modelled inside the thread frames by the static scheduling of the thread nodes. The method described

above wherein said designing step comprises the step of partitioning said at least partially scheduled description over a plurality of processors. The method describe above, wherein said designing step comprises the step of performing a hardware/software partitioning step on said at least partially scheduled description. The method described above, wherein before generating said plurality of static schedulings for at least one thread frame a task concurrency improving transformation on said thread frame is performed.

[0016]   It is a first aspect of the invention to disclose a method for designing an essentially digital system, wherein Pareto-based task concurrency optimization is performed. Within said method a system-level description of the functionality and timing of said digital system, said system-level description comprising of a plurality of tasks, is generated. Task concurrency optimization is performed on said system-level description, thereby obtaining a task concurrency optimized system-level description, including Pareto-like task optimization information. Finally said essentially digital system is designed based on said task concurrency optimized system-level description.

[0017]   It is a first embodiment of said first aspect to work with a grey-box description within said method.

[0018]   It is a second embodiment of said first aspect to include within said task concurrency optimized system-level description a real-time operating system, exploiting said Pareto-like task optimization information.

[0019]   It is a second aspect of the invention to disclose a method for designing an essentially digital system, wherein grey-box task concurrency optimization is performed. Within said method a description of the functionality and timing of said digital system is generated, said description being a grey-box system-level description comprising of a plurality of tasks. Task concurrency optimization is performed on said grey-box system-level description, thereby obtaining a task concurrency optimized grey-box system-level description. Finally said essentially digital system is designed based on said task concurrency optimized grey-box system-level description.

[0020]   It is a first embodiment of said second aspect to work with a Pareto-based approach for optimization.

9

[0021] It is a second embodiment of said second aspect to include within said task concurrency optimized system-level description a real-time operating system.

[0022] It is an embodiment of said two aspects to use a grey-box description wherein non-deterministic behavior of said digital system is modeled by interacting of said tasks, while each of said tasks describe part of the deterministic behavior of said digital system.

[0023] It is an embodiment of said two aspects to use within said task concurrency optimization a design-time intra-task scheduling performing design-time intra-task scheduling for at least two of said tasks separately, thereby generating a plurality of intra-task schedules for each of said tasks. Said design-time intra-task scheduling is also denoted a first static scheduling.

[0024] In a further embodiment thereof said plurality of intra-task schedules is a subset of all possible intra-task schedules, said subset containing essentially only Pareto optimal schedules. This means that in terms of cost - constraint terms, these schedules are withheld with for a certain cost the minimal constraint and for a certain constraint minimal cost. The cost can be the power consumption needed while executing a task on said digital device or an estimate thereof. The constraint can be the time needed for execution of a task on said digital device or an estimate thereof. Said time is also denoted the cycle budget.

[0025] In a still further embodiment said digital system is assumed to have at least one processor and said design-time intra-task scheduling takes into account processor power consumption optimization.

[0026] In an alternative still further embodiment said digital system is assumed to comprise of a plurality of processors, said design-time intra-task scheduling takes into account processor power consumption optimization, including assignment of tasks to processors.

[0027] Hence said task concurrency optimization, and in particular the static or design-time scheduling part thereof, performs a pre-ordering within tasks under real-time constraints while minimizing processor operation costs.

[0028] In a particular implementation said static scheduling exploits a mixed integer linear programming (MILP) approach.

[0029]   In an embodiment of these last two embodiments at least one processor is considered to be a multi supply voltage processor, with either fixed or variable supply voltages.

[0030]   It is an embodiment of said two aspects of the invention to design after static scheduling a run-time scheduler, being part of a real-time operating system. Said scheduler is capable of performing dynamic scheduling of at least two of said tasks, said dynamic scheduling comprising of selection for at least one of said tasks one of said generated static schedules. Said dynamic scheduling selection is based on cost-constraint information of said generated static schedules. Hence said dynamic scheduling comprises of a second time-constraint cost optimization problem. Said second time-constraint cost optimization optimizes a cost function comprising at least energy consumption of a processor, executing a scheduled task.

[0031]   In a particular implementation said dynamic scheduling exploits a constraint integer linear programming approach.

[0032]   Within said two aspects of the invention the concept of tasks is defined as grouping of logical operations, functionally grouped together. With task is meant a real-time task, thus at least partly dynamically created or deleted. Said tasks are typically concurrent, interacting or communicating. A suitable grey-box description can be based on a MTG-CDFG model, wherein the multi-thread graph (MTG) is used for inter-task descriptions and the control data flow graph (CDFG) is used for intra-task descriptions. Said tasks concurrency optimization taking into account only part of the logical operations needed for describing the functionality and timing of said digital system. Said tasks are design-time analyzable parts of said grey-box description.

[0033]   In an embodiment of the invention said grey-box system-level description comprise of a set of at least two or a plurality of tasks, also denoted thread frames, each of said thread frames being a set of at least two or a plurality of thread nodes. Said thread frame's or task's nodes comprise of logical operations, functionally belonging together. ·

[0034]   Within such frame-node description said static or design-time scheduling generates a plurality of static schedulings of at least two of the thread nodes of one of said thread frames. When said essentially digital system is modeled as a set of or a plurality of

11

processors, said generating of a plurality of static schedulings within a task comprising solving of a first time-constraint cost optimization problem wherein thread nodes are assigned to said processors and said cost essentially being related to said processors. Cost-cycle budget trade-off curves are generated for at least two thread frames, said cost-cycle budget trade-off curves comprise of a list of points representing part of all possible said static schedulings, said part being Pareto optimal schedulings in terms of the cost and cycle budget. Said generating of static schedulings for a thread frame can further be capable of modifying the amount of thread nodes within said thread frame.

[0035]   The invented design method is a system-level design approach, meaning that said task concurrency optimization is performed before hardware/software co-design, hence before it is decided which task will be performed by dedicated hardware or be mapped on a programmable processor. Thus a final hardware/software co-design step follows after task concurrency optimization. Said co-design step will result in a final implementation, whereon a real-time operating system, including a run-time scheduler, using Pareto-like task optimization information for his scheduling activities.

[0036]   In an embodiment of the invention wherein said grey-box description is obtained by performing task concurrency enabling steps on a first initial description or global specification of said digital system. Said task concurrency enabling steps include automatic grey-box description extraction, obtaining a first initial grey-box description or unified specification. Said task concurrency enabling steps include applying task concurrency improving transformations on said first initial grey-box description or unified specification, thereby obtaining said grey-box description being a task concurrency improved or enabled grey-box description.

[0037]   In an alternative implementation before said task concurrency optimization dynamic memory management and/or task level DTSE transformations are performed.

Brief description of the drawings

List of Figures

List of Tables

Detailed description of the invention

[0038]   We will first give a broad overview of the steps that we have invented to solve the problems presented in the prior-art. Afterwards we will explain in more detail each of the steps: grey-box model extraction, transformations, static and dynamic scheduling.

Overview

Motivation

[0039]   From Figure 1A, one can conclude that there is a lack of methodology and tool support at the system level for the *co-design of hardware and software* and for the *task concurrency management*, often resulting in an iterative and error-prone design cycle.

[0040]   At the top there is a need for an unified *specification model* with the power to represent system-level abstraction like *task concurrency, dynamic task creation, inter-task communication and synchronization*, and *real-time constraints*. The main problem is to close the gap from *concurrent, communicating task specification* to the *actual (usually single-threaded) target processor implementation*, without making a compromise on the required real-time performance and with actively considering cost effectiveness (especially energy consumption). A systematic approach towards (software) system design aiming at reducing the design-time and the number of errors and the debugging effort of embedded systems/software, consisting more and more of distributed (software) behavior running and communicating across multiple processors, is mandatory.

Target application domain

[0041]   With reference to Figure 1b, the target application domain of the task-level system synthesis approach advocated by us is advanced real-time multi-media and information processing (RMP) systems, such as consumer multi-media electronics and personal communication systems. These applications belong to the class of real-time embedded systems involving a combination of complex data- and control-flow where complex data types are manipulated and transferred. Most of these applications are compact and portable devices, putting stringent constraints on the degree of integration (i.e. chip area)

and on their power consumption. Secondly, these systems are extremely *heterogeneous in nature* and combine high performance data processing (e.g. data processing on transmission data input) as well as slow rate control processing (e.g. system control functions), synchronous as well as asynchronous parts, analog versus digital, and so on. Thirdly, time-to-market has become a critical factor in the design phase. Fourthly, these systems are subjected to stringent real-time constraints (both hard and soft deadlines are present), complicating their implementation considerably.

[0042]   Another major driver for this type of research are applications like the new MPEG4 standard (see fig. 12), which involves a massive amount of specification code (more than 100K lines of high-level C++ code) and which combines video, synthetic images, audio and speech modules with a system protocol. The IM1 application on which we are focussing is based on the MPEG4 standard, which specifies a system for the communication of interactive audio-visual scenes composed of objects.

Task concurrency management at the grey-box level

[0043]   We propose to look at a new abstraction level of mapping such real-time embedded systems, with emphasis on performance and timing aspects (concurrency management and meeting of the timing constraints or *'timeliness'*) while minimizing (abstract) processor cost overhead. Our focus lies especially on the task-level abstraction. But in contrast to existing approaches we will work neither at the detailed "white box" task model where all the operations are considered already during the mapping and where too much information is present to allow a thorough exploration, nor at the "black box" task model, where insufficient information is available to accurately steer even the most crucial cost trade-offs. Indeed, in the latter case the tasks are precharacterized by an initial design stage. But during the mapping, these costs and constraint parameters are not updated, which is crucial to obtain a good mapping involving transformations on the internal and I/O behaviour of the tasks.

[0044]   At our targeted abstraction level the distinction between "hardware synthesis" and "software mapping/synthesis" is practically gone so that all steps in the proposed design trajectory can then be shared for both target types of platforms. That enables

a much more global design space exploration than in the traditional way. One of the major consequences is that the system cost can be significantly reduced and tighter timing/performance constraints can be met. In addition, the automation support can be mostly unified, which heavily limits the man-power investment required in research, development and production at this stage.

[0045] The main problems are:

[0046] 1. Very complex (software type) specs. So the traditional solutions (white box view approach) fail. Instead we propose to use a grey-box modeling approach where only the essential data types, control/data-flow and dynamic constructs like semaphores and external events are present (see fig. 6). A real-life example is presented in Fig. 7 On this initial grey-box model, we have to apply task-level transformations to further reduce the complexity and to expose more concurrency.

[0047] 2. These systems are extremely dynamic due to the inherent user interactivity, the scalability and the presence of agents/applets. So conventional DSP models (derived in the past mostly from the "physical/link" layer of terminals) are not feasible any longer, and these have to be substituted by support of the global dynamic control flow at the grey box level. Instead, we need a model like the multi-tread graph (MTG).

[0048] 3. Many complex data co-exist where manipulation is more important than the applied operations. Hence a strong focus on storage and transfer is needed. Solutions from the operation-level can be largely reused here for the data transfer and storage exploration stage. However, in addition, aggressive static data type refinement and (due to the dynamic nature) dynamic memory management have to be supported. At the end the address and condition/loop overhead introduced by the code transformations of these stages has to be removed again too.

[0049] 4. Complex timing and concurrency constraints are present. As a result, in current design practice, nobody can really manage a thorough "exploration" at this stage either because of a too low abstraction level (white box level with too many details) or of not enough info (UML and black box models). In contrast, our grey-box abstraction provides the right level to efficiently explore this, i.e. focussing on the dynamic constructs (events,

19

semphares, threads) and the timing issues, together with the main data types and their control-data flow dependencies.

[0050] 5.   Cost-efficiency is still required for embedded systems, so power, required band-width, memory size are all very crucial to incorporate during the system design. Hence, we need a global exploration, driven by analysis feedback as opposed to the (nearly) blind mapping which designers are stuck with today (because only the last phase of the system design trajectory is supported by compiler tools). For this analysis fully automated and fast exploration tools with appropriate internal models are required, because interaction with designer is hopeless due to bookkeeping and global information processing during this "search space exploration."

Embedded systems - task-level design problems

[0051]   We aim at providing design support for the development of real-time embedded systems, with special focus at the task-level, so the understanding of the development process is essential.

[0052]   The cost-effective implementation of real-time embedded systems should span two levels of abstraction (see also fig. 8). At the more traditional 'operation/instruction level,' the design consists of the actual coding of the different concurrent sub-behaviours in the system, often called 'tasks', on the (programmable) instruction-set or custom processor. At a higher level, which we call 'task level,' a concurrency management stage must be performed. This stage takes care that the different tasks coordinate (i.e. communicate, synchronize, coordination of resources, . . .) on the same processor or on a set of processors, within the real-time constraints.

[0053]   At each concurrency level, decisions relevant to that specific abstraction-level are introduced. These decisions are propagated as top-down 'constraints' to the subsequent levels. This gradually restricts the design search space and supports 'gradual refinement' of the design across different abstraction levels, leading to a global optimal result. To cut the dependency on lower level design decisions – i.e. a bottom-up dependency – of a higher level stage, 'estimators' must be resided to which give an approximation for the lower level design decision impact. Similarly, bottom-up information regarding pre-taken

architectural (or other) design decisions (e.g. selection upfront the target programmable processor) needs to be propagated from the lower to higher level stages. The data concurrency level – which can be exploited by allocating 'array-processors in the architecture – will not be discussed further here. At each concurrency level, both the *data transfer and storage issues* and the *concurrency issues* are addressed. The former are performed in the *'Data Transfer and Storage Exploration'* (DTSE) stages (see fig. 8). These DTSE stages are all focused on the manipulation of the complex data types (like arrays).

[0054]   **Operation/instruction level.** This belongs to the prior-art and hence it will not be discussed here any further.

[0055]   **Task level.** The challenge of tomorrow is to map a complete system-level behavioural description, merely consisting of a number of interacting processes rather than a single algorithm to these valuable components.

[0056]   The main problem is to close the gap between these two abstraction levels (i.e. system-level versus operation/instruction-level) by automating the *'system/software embedding'* step; by this step, we mean the actions taking care that the set of processes can run on the same target by coordinating their interaction with each other and with the environment, and that the component is embedded in the rest of the system.

[0057]   A major feature of our proposed approach is that additional abstraction levels and design stages (labeled *'validation - refinement - optimization'*) are introduced. These have to be tackled *before* the traditional SW/HW partitioning is performed, as indicated in fig. 8.

[0058]   The main sub-problems which we discern and that should be solved at the task-level are:

[0059] 1.   **task concurrency extraction:** the system specification may contain implicit task-level concurrency, requiring a separate pre-processing step which analyzes and extracts the amount of concurrency. In several cases, (control) transformations provide considerable improvements on the available concurrency, the freedom in scheduling and the reduction of the critical paths.

[0060] 2.   **real-time system model:** powerful and portable programming models supporting real-time are needed which isolate the (task-level) specification from the

underlying processor hardware (either programmable or custom) and in this way enhance **portability** and **modularity** and stimulate reuse.

[0061] 3. **automatic time-driven task scheduling and scheduling algorithm selection:** as indicated above, the scheduling in the presence of various constraints is too difficult to perform manually. Moreover, there is a clear need for algorithms which consider the concept of time directly (i.e. 'time-driven'). The automated selection of the scheduling algorithm is needed since the requirements of different applications widely vary. A massive amount of work on scheduling has been performed and in this in a variety of communities but few approaches are available on 'selection.'

[0062] 4. **processor allocation and assignment:** as multi-processor targets are key elements in future systems, the partitioning of the specification over the different processors is too complex to be left to the designer, due to the complex and interacting cost functions, like schedulability, communication and memory overhead.

[0063] 5. **resource estimators:** in order to generate an optimal mapping and scheduling of the specification onto the available processors and in order to generate the schedule as well as to verify its feasibility, data on the use of CPU resource (i.e. cycles) and memory is required. Furthermore, the number of processors, buffer lengths, . . . are key to both scheduling and partitioning sub-tasks. Estimations can be generated using profiling, simulation, source code analysis and generated code inspection (e.g. linker map tables).

[0064] 6. **high-level timing estimators:** at a high-level in the design script, timing estimators are needed to guide the design decisions (e.g. processor allocation), ensuring timeliness of their outcome. It is merely estimators for the execution times of behaviour capable of dealing with partially implemented behaviour which are lacking rather than the timing analysis. Existing timing analysis approaches can be re-used by feeding in the execution time estimations provided by these tools. These estimators must be capable of dealing with, amongst others, incompletely refined communication (e.g. no insertion of data type conversion, protocol conversion, and even without bus assignment), absence of memory (hierarchy) decisions (e.g. levels of caching, number of memories and their number of ports, memory organization) and absence of partitioning and allocation decisions.

[0065] 7.   **interface   refinement:**   comprises   of   the   following: (1) synchronization refinement, (2) data type conversion, and (3) scalar buffering. The first refines the high-level communication present in the input specification into a concrete IO-scenario taking into account processor characteristics (like peripherals), the throughput requirements and the size of inserted buffers. Software device drivers have to be synthesized and dedicated logic must be inserted to interface the (programmable or custom) processors to each other or to dedicated peripherals. The second ensures data type compatibility on the different communication sides by inserting behavioural conversions.

Pareto curves and their use at the task level

[0066]   Due to the task-level exploration with the steps proposed above, it is possible to derive Pareto curves which indicate the points in the search space where a (set of) cost(s) is optimal relative to a constraint like the time budget. Non-optimal points in the search space that are "dominated" by other points should not be considered for subsequent implementation. That information is crucial for the system designer to decide on trade-offs with other subsystems present in the entire application. If the cost is energy, and when we map the initial MTG/CDFG specification on a pair of two processors, one with high Vdd and one with low Vdd, we can trade-off time and energy by moving more tasks from the low Vdd to the high Vdd one. The execution time indeed goes up roughly linearly with Vdd while the energy decreases quadratically. The optimal trade-off points typically lead to a curve as indicated in fig. 9. That curve should include both the data transfer and storage (DTS) related cost and the processor core related cost. In practically all current exploration approaches that DTS cost is ignored or considered as constant. However, when the time budget is decreased, both these costs typically go up. The DTS cost increases because usually more system pipelining (buffering) has to be introduced and also the mapping on the SDRAM organisation and the processor caches becomes more costly under stringent time budget constraints. The processor cost increases because more subtasks have to be moved from the low Vdd to the high Vdd processor during assignment and scheduling, in order to meet the more stringent timing requirements.

23

[0067]   Due to the aggressive application of task-level concurrency extraction and concurrency improving transformations, the additional concurrency and reduced complexity can be exploited to move the location of the Pareto curve left and down for the processor core contribution (see fig. 10). This indeed allows to better exploit the low Vdd processor during the task-level assignment and scheduling. Consequently the (energy) cost goes down. The DTS cost can be improved also because DTSE related code transformations allow to remove redundant accesses, and to improve the regularity, locality and data reuse behaviour. In addition, the concurrency improving transformations provide more access ordering freedom to remove part of the internal overhead (size especially) for the dynamically sized buffers. The transformations can however also be used to speed up the implementation, i.e. to break up the critical path and hence enable tighter time budgets to be achieved (at a potentially higher cost of course). As a result, the global Pareto curve after the transformations represents significantly improved solutions in the search space, both in terms of the cost required for a given time budget and in terms of the minimal time budget that can be reached.

[0068]   When these Pareto curves are precomputed for several task clusters at design time for a given embedded application running on a given processor configuration, they can then be used also to improve the global behaviour of the system in a very dynamic environment. Indeed, the steering of the scheduling and processor assignment of these task clusters on a set of processor and storage resources (a "platform") can then be dynamically improved in the presence of "foreign" tasks that are entered on the platform as agents or applets (see New Task3 in fig. 11). The cost-time behaviour (the Pareto curve) of these entering tasks will then typically not be known, but their required time budget is almost always derivable (or estimatable). The required extra time on the resources then has to be subtracted from the currently running task clusters (Task1 and Task2 in fig. 11) and this involves an optimisation problem because depending on how much each of them is squeezed, the cost effect will be lower for one/some of them compared to the others. For instance, in the example of fig. 11 it is clear that when the initial time budgets of Task1 and Task2 are to the right side of TB1, it is best to squeeze Task2. As soon as that point is reached, it is better to start squeezing Task1 until it cannot be speeded up further. Such trade-offs to be taken can be effectively taken into account at run-time (by a customized part of the embedded operating

system) by analyzing their precomputed Pareto curves. This will allow to significantly decrease the energy or size costs while still meeting all the timing constraints, because each of the task clusters can operate close to one of its optimal points in the overall search space.

Extraction of the grey-box model

[0069] The C++ code is considered as a description of tasks performing specific logical operations. These tasks consist in their turn of threads that run continuously and independently from each other. The threads are a CDF description of functions triggered by an event signal. The primitive semantic that forms the thread is a thread node. This three-layer approach is described further in detail. In the lower level, the CDF parts of the functions are split in "seen" and "hidden" sublayers. The lowest (hidden) sublayer is a group of thread nodes consisting of CDF with purely primitive operations. In the highest (seen) sublayer, these thread nodes are grouped in threads that also include CDF with the thread nodes as primitive. The distinction between a thread and a thread node is the dynamic, non-deterministic behavior of the former in comparison to the primitive operations performed by the latter. In the middle level the "seen" and "hidden" approach is again applied. The dynamic threads are connected with MTG semantics and grouped in higher-level tasks. The criterion for this grouping is the logical functionality of the system and therefore could change after the transformation step. Since the main functionality remains the same and the general steps are called again and again in the process, a global view of the system can be generated. In that sense a grey-box model can be extracted both for the MTG (middle) and the CDF (lower) layers. The steps to be performed to get from the initial system specification (C/C++ code or high level information) to the grey-box model to be used as input for TCM exploration are the following:

- [0070] Extraction of the black-box model based on the high-level system specification, which is in most cases available. The global view of the system's functionality is represented in black boxes with no information about their content, the algorithms used, the operations performed on the data types or the detailed dependencies between them. It is only a very abstract representation of

25

the system to give the general picture of the function boundaries that must be retained during the changes later in the exploration process.

- **[0071]** Looking inside the black-boxes, the operations of the main functionality are drawn with the function scope of the C/C++ code. The functions are considered as thread nodes for the moment and the black-box approach is extended with control and data flow edges to connect these thread nodes.

- **[0072]** Operations like variable assignment or primitive functions are grouped together in a thread node because they are considered as an implementation of an abstract CDFG functionality.

- **[0073]** Conditional guards of branches that are not local in a function but depend on a completely different functionality indicate an ordering or a dependency between the two threads. From this point on, the top-down approach is taken, so we work on the details of the functions but always keeping the global picture of the system. The "for" and "while" loops are detected and represented inside the function scope. They are considered as important if the number of iterations is data dependent or if the loop body times the number of loop iterations is large and hence represents a high execution time on the final target processor.

- **[0074]** Non-deterministic behavior that has to do with synchronization is taken into account. Semaphore reserve and release indicate the boundary of a group of functionalities interacting with another one as far as data interchange is concerned. Other non-deterministic issues like the creation of threads or/and tasks are also annotated on the diagrams inside the function scope.

Task-level transformations

[0075]   In this section we will present first the real-life results of task-level transformations. Afterwards, we will give an overview of the different types of transformations that we distinguish.

Real-life examples of transformations that improve the amount of concurrency
Description of the transformations

26

[0076] When developing an initial model of a system, specification engineers normally spend neither time nor energy to expose all possible concurrent behavior. They are mainly concerned with the functionality and the correctness of the system. As a result, the software model that they create hides almost all concurrency. However, for the design engineers it becomes an almost unmanageable and very time consuming problem to extract the concurrency from this initial specification and to map it on a concurrent architecture within stringent time constraints. Therefore, they will either redesign the system from scratch or use a faster and thus more power consuming architecture. In our design flow, on the contrary, we model the initial specification on a grey-box level that allows us to systematically identify and transform the time critical parts. We start to analyze the data dependencies that inhibit concurrency. When no recursion is involved, the scope of these data dependencies can be reduced by adding extra buffering (and possibly a small additional routine to synchronize the data). However, the main difficulty is to detect these data dependencies. Software designers often prefer to use run-time constructs to model data dependencies between the boundaries of tasks as this gives them more flexibility to change their initial specification afterwards. Hence, they conceal data dependencies with semaphores or mutexes, seemingly required for a non-deterministic behavior. Therefore, we carefully analyze all these constructs in our grey-box model to determine the really needed ones. This is an enabling step that to allow us to then actually break dependencies by introducing a small buffer. We will illustrate this approach with two real examples extracted from the IM1 player. To make the examples comprehensible, we will give here a short introduction of the parts that we have invested.

[0077] The IM1 player is based on the MPEG4 standard, which specifies a system for the communication of interactive audio-visual scenes composed of objects. The player can be partioned in four layers (see fig. 12): the delivery layer, the synchronization layer, the compression layer and the composition layer. The delivery layer is generic mechanism that conveys streaming data to the player. Input streams flow through the different layers and are gradually interpreted. We mainly focus on the compression layer. This layer contains several decoders. Two important ones are the BIFS and the OD decoder. The BIFS decoder interprets the data packets that describe the composition of the scene objects. The output of

this decoder is a graph. Each node in this graph presents either an object in the scene or describes the relation between the different objects. However, to represent some of the scene objects extra AV-data is needed. How this data needs to be conveyed to the compression layer and how it needs to be decoded is described in Object Descriptors (OD). These ODs are encoded in one or more streams entering the compression layer and are decoded by the OD decoder. The ODs are linked with the nodes requiring the extra AV-data.

[0078] In the first example, we have changed the relation between the synchronization and compression layer. Originally, the two tasks were separated with a buffer protected against simultaneous access with a mutex, i.e. WaitForData mutex presented in fig. 13 left. This figure is a part of the grey-box model that we have used for our analysis. As a result of this semaphore, both tasks were always executed sequentially. However, this semaphore was only needed to model a data dependency: the compression layer cannot start to decode before data is available at the output of the synchronization layer. Therefore, we have replaced the semaphore with a data dependency, indicated in the figure with the bold dashed dotted line. Moreover, no recursion was involved with this data dependency. Hence, by increasing the size of the (small) buffer, the synchronization layer is allowed to handle a next data packet, while the compression layer is decoding the previous one. At the cost of some extra buffering, we have succeeded in increasing the amount of concurrency.

[0079] A second example illustrates a similar principle. In the compression layer of the player, the OD and BIFS decoder share a data structure, i.e. the graph representing the scene. Above we have described that this graph is the output of the BIFS decoder. The OD decoder also accesses this graph to link ODs with the nodes in the graph. The graph is protected against simultaneous access by the BIFS and OD decoder with a semaphore. As a result, they cannot run in parallel. After a more thorough investigation, we have found out that the links between the ODs and the nodes in the graph are not needed during the execution of either the BIFS or the OD decoder. Therefore, we have broken the data dependency by postponing its initialization until after the execution of both decoders. Hence, the OD decoder does not need to access the graph and can work independently of the BIFS decoder (see fig. 13 right). This again costs some extra buffering plus a small extra initialization routine.

[0080] As a side-effect of this transformation, extra concurrency inside the OD decoder becomes available. The ODs were previously decoded sequentially because of the access to the shared graph. By removing this dependency, they can now be executed in parallel. Using this concurrency we create extra freedom for the task scheduling.

[0081] We believe that such transformations can be generalized for similar graphs with other contents of task behavior. In the next section we will show how these transformations affect the overall cost-performance trade-off of the system even on a weakly parallel platform.

Results of the transformations: task scheduling experiment on a weakly parallel platform

[0082] Two parallel ARM7TDMI processors, running at 1V and 3.3V respectively, are used to illustrate the scheduling. The main constraint for scheduling is the *time-budget*. For one task, we use t_L and t_H to denote the its execution time on the 1V ARM7TDMI and 3.3V ARM7TDMI respectively and Δt to denote the difference between these two execution times. Similarly, we use E_L and E_H to denote its energy cost of the processor to run on 1V ARM7TDMI and 3.3 ARM7TDMI and ΔE to denote the energy difference. Our calculations are based on the following formulae.

$$\Delta t = t_L - t_H = (1 - \frac{1}{3.3})t_L \tag{6.1}$$

$$\Delta E = E_H - E_L = (3.3^2 - 1)E_L \tag{6.2}$$

[0083] For the first experiment the time-budget starts at the reading of one VOP[1] data and expires at the end of the decoding of the data (see fig. 12). The original implementation dictates that the data can only be read and decoded sequentially. Due to this limitation, we can only reduce the period by executing some tasks on a high Vdd processor.

[0084] The second experiment gives similar results. Given is a group of media objects, i.e. a group of nodes. The time-budget starts now at the beginning of parsing the first node and expires at the end of the setting up the decoder for the last node. Each of these nodes will undergo three steps: first its description is parsed by the BIFS decoder, the related OD is then parsed by the OD decoder and finally the AV decoder described in the OD is set

up by the OD decoder. Originally the BIFS and the OD decoder run sequentially. That is, OD decoding only starts after all the nodes finishes their BIFS decoding. The time-budget and total energy cost of the original Pareto-optimal points in the search space are shown with asterix points for both the first and second experiment on the left and right side of fig. 34 respectively.

[0085]   After the TCG related transformations discussed above that lead to the transformed task graph, we can remove these implicit sequential restrictions in the code. Consequently, we can better utilize the two processors by exploiting the concurrency. Hence, we can achieve the same time-budget with less energy cost or we can reduce the minimally achievable time-budget due to the added flexibility. The scheduling results for the transformed graphs are also shown in fig. 34, with the diamond points. Gains of 20% and more are achievable in energy cost and the maximal speed can be increased with nearly a factor 2.

Overview of different transformations

Simplify the grey-box model

•   Hierarchical rewriting

[0086]   In the concept of focusing in the TCM related parts, the code that is not providing any freedom to the transformation and scheduling steps is hidden. This is done in two layers of hierarchy according to the following criteria:

-         The critical-path: the control flow and data flow of the data types that are relevant to the system main functionality.

-         The most important data types of the system and the dependencies on them like loops whose number of iterations is data dependent.

-         The existence of dynamic events that include the semaphores and the mutex, the dynamic creation of tasks and the event signals between the environment and the system.

-         The timing information: this includes the average execution time of the functions and the execution time of a loop. The loops are important because of the fact that

---

[1] Video Object Plane

even if the loop body is not time consuming, the number of iterations may be large enough to make the total execution time of the loop "interesting." In the case where the white box specification is not available, the time information is being derived by the partial system specification or by estimates based on logical assumptions that are being verified later in the design process. Examples illustrating the transformation are in Fig. 15 and Fig. 16.

[0087] In the lower layer the CDF parts of the functions are "seen" or "hidden." The outcome at this level is thread nodes consisting of CDF of purely primitive operations. In the middle layer these thread nodes are grouped in threads that also include CDF with the thread nodes as primitive. In this layer the "seen" and "hidden" approach is again applied. Since the main functionality remains the same and the general steps are called again and again in the process, a global view of the system can be generated. In that sense a grey box model can be extracted. In each exploration step, we focus first locally in some parts that include the basic bottlenecks (see criteria below) and "hide" the rest threads in black boxes. The semantics of the MTG model are expanded for this two-layer distinction. For the "hide" parts in both layers a table of correspondence is used. The parts of a function body that are not relevant to the CDF and to the main functionality of the system are grouped and given a general name indicating the "functionality" that is performed. A function bubble is used for their representation in the TCG. The other parts of the function body that are important according to the criteria are kept in the "seen" part. In this way there is less details in the diagrams, a more global view of the functionality of the system and the complicated but unnecessary information is pruned away. On the other hand if later in the process we find that it is needed to split the "bubble" (because it takes considerable amount of time or because we need to take a closer look at the CDF) we still have the "lighter-grey" box model by using a table.

- Hide undesired constructs

[0088] In this step we "hide" the parts of the graphs that introduce complexity without adding any new TCM related concepts. This "hiding" is done without trade-off because at this pre-processing level the transformation should just simplify the initial TCG. In that context, the transformations should not affect the CDF of the system. The functionality remains the same but parts that are not related to the TCM focus are pruned to the "hidden" 3rd layer of DTSE. Example: The ODDecoder is creating a FileReader thread

31

if mActive is not set in the RequestChannels and the m_state is required. Illustrated in Fig. 17.

Create freedom

- Code expansion

[0089]   Example: a subroutine that is called by different threads and a part of it is "hidden" but the "seen" part is instantiated in the different calls to allow more freedom for the scheduling later on. There is however a more complex case where the "seen" part changes according to the context of the thread from which it is called. This can be achieved by passing different argument values or by setting different variable values. In this case, the "seen" part consists of two different instances according to the context. Each instance will be used in the corresponding thread. Examples illustrated in Fig. 18, Fig. 19 and in Fig. 20.

[0090]   It should be stressed here, that this kind of transformations do not affect the functionality of the system but only give more freedom to scheduling by reducing the complexity in the diagrams.

- [0091] Remove constructs that prohibit concurrency analysis

-    DTSE objective: The general concept is to propone the reading and postpone the writing.

-    TCM objective: the basic goal is to introduce as much concurrency as possible. In this context the parts of the system that are independent but executed sequentially because of the nature of C++, are being decoupled by the control flow and considered to be executed in parallel.

[0092]   Example follows in Fig. 21:

- [0093] Transform constructs that prohibit concurrency analysis

-    Trade-off: If the transformed graph is not equivalent to the initial one, the new degree of complexity and concurrency in compare to freedom should be taken into account. The control edges that impose strict constraints to TGM are moved or an assumption is introduced to create more freedom. An example of this transformation is given in section 6.3.1. Example follows in Fig. 22 and in Fig. 23

- No trade-off: At this step only the transformations that result in the same functionality are performed. The functions are split into these thread nodes that are common between them and then merged into one. Example follows in Fig. 24.

Concurrency analysis

[0094] The system is analyzed globally in order to detect the parts that can run independently and thus concurrently. This step focuses on the potential parallelism and lists the bottlenecks that inhibit it. This is illustrated in Fig. 25.

Reduce complexity exposed by more detailed concurrency analysis

- Remove unused code

[0095] This step aims at keeping only the main operations of the system functionality and to remove the redundant code as far as the asserting (extra) checks and the variable-set are concerned. This is similar to step 2 for creating freedom but in this phase, the parts removed are those that were not obvious in the plane C++ code. Because of the concurrency analysis information (including data flow arrows) the redundant information is already abstracted away, the system functionality is handled more globally, allowing for extra redundant dependency/constraint/code detection. E.g. when the value of an argument (important data), which is passed from one function or instance of a class to another, is known in advance, only the body of the branch that is triggered depending on this value is left in the diagrams. Example follows in Fig. 26 :

- Reduce complexity by trading off globality of exploration and hence result quality Example follows in Fig. 27:

- Weight-based hiding of less crucial constructs

[0096] Hide the code that is still complex or even impossible to handle (i.e. non-determinism) and is less relevant in comparison to the other parts that are retained. The hiding is done by putting a threshold (complexity target based) on the list of retained code constructs, which are first ranked based on the relative importance of still difficult to handle issues.

- Partitioning

[0097]   At this phase the threads are grouped in task clusters that form the dark-grey-box model of the system. This is the third and higher level of hierarchy (see above). In the highest layer of the grey-box model, tasks are grouped into task clusters where the main characteristic is that low communication and interaction are present between the combined tasks. So, one objective is to decouple them in the highest possible degree e.g. for later task scheduling Typically also the dynamic move of sets of tasks over a network (e.g. as Java applets) is captured through the concept of a task cluster that is dynamically created at other nodes. Example follows in Fig. 28:

Scheduling at the task-level

[0098]   The first section presents how we organize scheduling at our abstraction level. Moreover, it illustrates the benefit that can be obtained using both our static and dynamic scheduling techniques on real-life example. The second section gives another illustration of the results that can be obtained with our static scheduling techniques.

Scheduling at the Task-level

[0099]   The design of concurrent real-time embedded systems, and embedded software in particular, is a difficult problem, which is hard to perform manually due to the complex consumer-producer relations, the presence of various timing constraints, the non-determinism in the specification and the sometimes tight interaction with the underlying hardware. Here we present a new cost-oriented approach to the problem of task scheduling on a set of processors. It fits in a global task concurrency management approach developed at IMEC. The approach uses as much as possible pre-ordering of the concurrent behavior under real-time constraints and minimizes the run-time overhead. At the same time, the scheduler will try to minimize cost such as the energy consumption. The approach is shown for two processors but it is easy to expand this method to include more than two PEs

Overview

[0100]   An embedded system can be specified at a grey-box abstraction level in a combined MTG-CDFG model. This specification is functional in representing the idea of concurrency, time constraints and interaction at either an abstract or a more detailed level,

depending on what is required to perform good exploration decisions afterwards. According to the IMEC TCM approach, the task concurrency management can be implemented in three steps. Firstly the concurrency extractionis performed. Task transformations on the specified MTG-CDFG will be applied to increase the opportunities for concurrency exploration and cost minimization. After the extraction, we will get a set of thread frames (TF), each of which consists of many thread nodes and which can be looked at as a more or less independent part of the whole task. Then static scheduling will be applied inside each TF at compile time, including a processor assignment decision in the case of multiple PEs. Finally, a dynamical scheduler will schedule these TFs at run time on the given platform.

[0101] The purpose of task concurrency management is to determine a cost-optimal constraint-driven scheduling, allocation and assignment of various tasks to one or more processors. In this report we consider a system consisting of two different processors, on which a thread node can be executed at different speed with different cost, here energy consumption. Given a thread frame(TF), static scheduling is done at compile time to explore the possible points on the global Pareto curve, where each point means a different choice in cycle budget and energy consumption trade-off. The idea here is that each thread node in the TF can be allocated to either of the two processors at different relative order if such an assignment can satisfy all the time constraints internal or external to that TF. By doing this, the cycle budget and energy consumption for the while thread frame will also change accordingly. Each of these points is a different choice of all the possible allocation and scheduling options for this TF and the lowest edge of all these points is just the Pareto curve. Since the static scheduling is done at compile time, computation efforts can be paid as much as needed, provided that it can give a better scheduling result and reduce the computation efforts of dynamic scheduling.

[0102] At run time, the dynamic scheduler will then work at the granularity of thread frames. Whenever new thread frames come, the dynamic scheduler will try to schedule them to satisfy their time constraints and minimize the system energy consumption as well. The details inside a thread frame, like the execution time or data dependency of each thread node, can remain invisible to the dynamic scheduler and reduces its complexity significantly. Only some useful features of the Pareto curve will be passed to the dynamic

scheduler by the static scheduling results, and be used to find a reasonable cycle budget distribution for all the running thread frames.

[0103] We separate the task concurrency management into two separate phases, namely static and dynamic scheduling, for three reasons. First, it can better optimize the embedded software design. Second, it lends more run time flexibility to the whole system. Third, it minimizes the run time computation complexity. The static scheduler works at the grey-box level but still sees quite a lot of information from the global specification. The end result hides all the unnecessary details and the dynamic scheduler only operates on the granularity of thread frame, not single thread node.

Static scheduling

[0104] The behavior of a thread frame can be described by task graphs like Fig. 29, where each node represents what functions to perform and their performance requirements. Each edge represents the data dependency between these nodes. The task graph we use here can be seen as a simplified subset of the MTG model and each function is a thread node in MTG. This example is part of a real voice coder. Tab. 1 gives the performance of each node on our two processors. Here some assumptions similar to those in the later ADSL example are made. Though it is not showed here, we can add some other time constraints to the task graph, for instance, task t_4 must start n time units after task t_0 ends. The TF is scheduled in a non-preemptive way here because we have the *a priori* knowledge of all the nodes involved.

[0105] We use a MILP algorithm to solve the static scheduling problem. Other algorithms can be used here as well. Before giving the algorithm, we need some variable definitions.

[0106] **Definition All variables used for static scheduling.**

L, H *two processors*

$$\delta_{i,L} \quad \begin{cases} 1 & task\ i\ is allocated\ to\ processor\ L, \\ 0 & otherwise \end{cases}$$

$$\delta_{,H} = \begin{cases} 1 & task\ i\ is\ allocated\ to\ processor\ H, \\ 0 & otherwise \end{cases}$$

$C_{i,L}$ *the execution time of task i on processor L*

36

$C_{i,H}$ *the execution time of task i on processor H*

$E_{i,L}$ *· the energy consumption of task i on processor L*

$E_{i,H}$ *the energy consumption of task i on processor H*

T_i^S *the start time of task i*

T_i^E *the end time of task i*

$b_{i1,i2} = \begin{cases} 1 & task\ i1\ ends\ before\ task\ i2\ starts, \\ 0 & otherwise \end{cases}$

[0107]   Our aim is to minimize the energy consumption of the whole thread frame, so the object function can be easily defined,

$$Minimize : \sum_i E_{i,H}\delta_{i,H} + E_{i,L}\delta_{i,L}. \tag{6.3}$$

[0108]   The following constraints have to be fulfilled.

[0109]   1. Mapping Constraints. Each thread node t_i is executed on only one processor, which is represented in Eq. 6.6.

$$\delta_{i,L} \leq 1 \tag{6.4}$$

$$\delta_{i,H} \leq 1 \tag{6.5}$$

$$\delta_{i,L} + \delta_{i,H} = 1 \tag{6.6}$$

[0110]   2. Timing Constraints. The task end time is the addition of task start time and execution time.

$$T_i^E = T_i^E + C_{i,L}\delta_{i,L} + C_{i,H}\delta_{i,H} \tag{6.7}$$

[0111]   3. Precedence Constraints. If thread node t_i is the immediate predecessor of thread node t_j, then

$$T_i^E \leq T_j^S \tag{6.8}$$

[0112]   4. Processor Sharing Constraints. For any two thread nodes t_i and t_j that are allocated to the same processor, if t_i is a predecessor or successor of t_j, it can never happen that one node will start before the other one ends due to the precedence constraints. Otherwise, the binary variable $b_{i,j}$ is used to describe the processor sharing situation.

$$T_i^E \leq T_j^S + (3 - b_{i,j} - \delta_{i,L} - \delta_{j,L}) * C_1 \tag{6.9}$$

$$T_j^E \leq T_i^S + (2 + b_{i,j} - \delta_{i,L} - \delta_{j,L}) * C_2 \qquad (6.10)$$

$$T_i^E \leq T_j^S + (3 - b_{i,j} - \delta_{i,H} - \delta_{j,H}) * C_1 \qquad (6.11)$$

$$T_j^E \leq T_i^S + (2 + b_{i,j} - \delta_{i,H} - \delta_{j,H}) * C_2 \qquad (6.12)$$

$$(6.13)$$

[0113] In the above equations, Eq. 6.9 and Eq. 6.10 are sharing constraints on processor L. When thread node i and j are both allocated to processor L, i.e., $\delta_{i,L} = \delta_{j,L} = 1$, if $b_{i,j}$ equals one, i.e., task i ends before task j starts, Eq. 6.9 gives the exactly same hard constraint $T_i^E \leq T_j^S$ and Eq. 6.10 becomes $T_j^E \leq T_i^S + C_2$, which can be ignored if C_2 is large enough; if $b_{i,j}$ equals zero, Eq. 6.10 gives the hard constraint $T_j^E \leq T_i^S$ and Eq. 6.9 can be ignored. Eq. 6.11 and Eq. 6.12 describe the similar sharing constraints on processor H. When node i and j are allocated to different processors, all of the above equations will be ignored. Constants C_1 and C_2 should be large numbers to ensure that the ignored equation will bring no effect.

[0114] At the end, a deadline for the whole thread frame can be given, which introduces a global constraint to all the nodes,

$$T_i^E \leq Deadline.$$

[0115] Now our problem becomes a deadline constrained energy minimization problem. To every given thread frame deadline, the above MILP equations will give an energy optimized solution. In fact, time constraints other than the whole thread frame deadline can be added easily. For instance, there can be a separate deadline for each node.

[0116] We have done the static scheduling for the thread frame shown in Fig. 29 with the performance given in Tab. 1. When the deadline of the whole thread frame is 100 time units, the result given by the MILP solver is shown in Fig. 30.

[0117] Changing the deadline for the whole thread frame, we can obtain a series of points. Each of these points is the optimized energy cost at the fixed deadline and they make up the well known Pareto curve in Fig. 9.

[0118] From these points, several ones will be chosen by the static scheduler as typical cases(those indicated by triangles) and will be passed to the dynamic scheduler. The

more points are passed, the better a result the dynamic scheduler can achieve, but at a higher run time computation complexity.

Dynamic scheduling.

[0119]   Static scheduling provides a series of possible options of allocation and scheduling inside one thread frame, but actually which option would be chosen is decided only by the dynamic scheduler at run time. The dynamic scheduler takes into consideration all the computation requests from all the ready to run thread frames and tries to find a choice for each thread frame so that the whole combinational energy consumption is optimal. At this stage, the clustered entities, namely thread frames, have already been distributed and assigned across the allocated processors. The dynamic scheduler operates on the base of thread frames and tries to satisfy the inter-task timing constraints.

[0120]   Take the two thread frames in Tab. 2 as instance. Each of these two thread frames has three options corresponding to different cycle budget and energy cost combination. These options are identified by the static scheduler. At run time, if the total cycle budget for these two thread frames is 100, the energy optimal scheduling is option 1 for thread frame 1 and option 3 for thread frame 2. However, the optimal scheduling will become option 2 for thread frame 1 and option 3 for thread frame 2 when the cycle budget is 140. Both cases are depicted in Fig. 31.

[0121]   Inter-task time constraints, such as the data dependency or execution order among the thread frames, can also be taken into account at this step.

Experiment On ADSL.

[0122]   In the last section, we have only illustrated the general method we proposed in the task concurrency management context. To better assess the viability of our approach, we have applied it to a more complex ADSL(Asynchronous Digital Subscriber Line) modem application.

ADSL system architecture.

[0123]   Fig. 33 shows the system architecture of the ADSL modem.

[0124]   This design consists of both hardware components and an important part of embedded software which exhibits real-time constraints. The digital hardware part includes two parallel datapaths for the receiver and transmitter respectively. The receiver

datapath consists of a front-end, a FFT transformer, a QAM demapper, an error and noise monitor and an error-correction decoder. The transmitter datapath has a similar structure. Both datapaths have their own hardware timing controller (DSTU=DMT Symbol Timing Unit). These DSTUs will activate the processors at the correct moments to do the appropriate processing of the DMT (Discrete Multitone) symbols.

[0125]   Next to the hardware components, an ARM core processor runs the software that is responsible for programming and configuring the hardware. It has a control part, that configures the hardware to execute the different stages of the initialization sequence, and an algorithmic part to execute DSP functionality not implemented in hardware. The control part of the embedded software can be described as a *reactive system*, reacting on events generated by the monitor and other hardware modules, taking into account real-time constraints imposed by the ADSL standard.

[0126]   In our example, we try to schedule the system consists of the two DSTU and SW modules (the modules in shadow) and model it into two threads. The two DSTU controller make up a thread frame which is generated periodically every 230μs and the deadline equals the period. The SW module generates another thread frame dynamically when the corresponding event has been triggered. The deadline for the SW thread frame is derived from the ADSL standard. The SW is executed serially, i.e., no new SW thread frame will be generated when there is another SW thread frame being processed. THence, at most two thread frames are executed at any time, one from the DSTU thread and one from the SW thread.

Assumptions.

[0127]   For the first step, we consider only two processors, one working at a higher voltage(V_{HIGH} = 3, 4 *or* 5 V) and the other working at a lower voltage(V_{LOW} = 1 V). The reason to use the two processor architecture is that it can provide a way to exploit the concurrency inherent in the tasks and consume less energy than the one processor architecture. Regarding the latter point, if there is only one processor, it has to be fast enough to handle the heavy load bursts, and a fast processor is typically also a power greedy processor. However, heavy load bursts come only occasionally. At other times, all the tasks will still be executed on that fast, power greedy processor, though it needs not to be so fast.

Even with modern power control technique that the fast processor can be shut down at idle time, it still consumes more energy than the set-up with the two processors working at different voltage, as we will prove later.

[0128] The maximal work frequency f_{max} of the processors can be computed as below,

$$t_{delay} \alpha \frac{CV_{dd}}{I} \alpha \frac{CV_{dd}}{(V_{dd} - V_{th})^2} = \frac{C}{V_{dd}},$$

$$\therefore f_{max} \alpha V_{dd}.$$

[0129] We assume also that the work frequency f is proportional to V_{dd}. In CMOS digital circuits, most of the power dissipation comes from dynamic power dissipation. For one time cycle, the average dynamic power dissipation can be computed as,

$$P_d \alpha V_{dd}^2 f \alpha V_{dd}^3.$$

[0130] For our experiment, we use the values in Tab. 3.

[0131] Other assumptions taken here are: the processor is powered down automatically when it is idle and no context switch overhead is considered in the current experiments.

[0132] The timer threads work at a rigid $230\mu s$ period and provide the needed control on the hardware, one for the transmitter and the other for the receiver. In the original design, the microcontroller of the timer will execute 128 instructions at most in one period, which is also its deadline. Most of the timer microcontroller instructions are testing, setting and resetting some signal line or register field, which can not be executed on a general processor like the ARM as efficiently as it is done on the specific microcontroller. Thus we assume each microcontroller instruction will take 5 general purpose processor cycles to execute it. Therefore, the worst case execution time of one timer is 64 μs on a 10MHz processor.

[0133] The SW thread is a sequence of sporadic tasks, or events. They are released by the hardware part and have to be completed by their deadline. Another issue worth noticing is that they are executed in a strict sequential order, one task will only start after the previous one ends and no overlap will happen between them. The deadlines and

41

execution times of these tasks are listed in Tab. 4, where the deadline is extracted from the initialization sequence of the ADSL standard and the execution time is measured on a 10MHz processor.

Algorithms.

[0134] Two threads are presented in this experiment, one consists of the two DSTU and the other is the SW controller. For the SW thread, a thread frame is generated dynamically and it has only one thread node. For the timer thread, two thread nodes are generated every symbol period. A thread frame can cover one, two or even more symbol periods and we call it a time block(TB). The more symbol periods in a timer thread frame, the less processor utility ratio we will get due to the increasing idle time. But also less computation efforts are needed at run time since there are less thread frames to be considered by the dynamic scheduler now. There is another difference from what we state in the last section. The execution time and deadline of the SW thread, varying from tens to hundreds of symbol periods, is much longer than the timer TF time granularity, which is only a few symbol periods. Therefore, one SW TF will have to be scheduled simultaneous with many timer TFs so the SW TF is intersected into many pieces by them. Accordingly we changed the dynamic scheduling method a little. When no SW event is triggered, the dynamic scheduler will choose the lest energy consumptive scheduling in which all the timer thread nodes are executed on the low voltage processor. Whenever a SW event is triggered, the dynamic scheduler will find out the execution order of all the TFs involved to save the energy consumption as much as possible.

[0135] We use (L,H) pairs to represent a scheduling decision. L is the set of thread nodes assigned to the low voltage processor; H is the set of thread nodes on the high voltage processor. For instance, (1,1)+(s,_) means one DSTU timer thread node and the SW thread are assigned to L, the other DSTU thread node is assigned to H at that TB.

[0136] To formalize the problem, suppose we have n static scheduling options for one time block. For each option, $C_{T,i}$ and $C_{S,i}(i = 1,2,K,n)$ represent the time that can be used to execute the timer and SW thread in one TB respectively. Similarly, $E_{T,i}$ and $E_{S,i}(i = 1,2,K,n)$ represent the energy consumption for timer and SW part in one TB. We will schedule with a time granularity of TB. Suppose the execution time of the coming SW

42

thread node is C and the deadline is D. If we have n possible choices for a TB and let $l_i(i = 1,2,\text{K},n)$ represent the number of TBs for each choice in the scheduling, to find a feasible scheduling is to find a set if l_i which can provide enough execution time for that task before the deadline meets. To find an optimal scheduling in energy cost is to choose one with the minimal energy consumption among these feasible schedulings. This can be restated as a constrained Integer Linear Programming(ILP) problem as below,

$$\sum_{i=1}^{n} l_i \cdot C_{S,i} \geq C, \tag{6.14}$$

$$\sum_{i=1}^{n} l_i \leq \frac{D}{TB}, \tag{6.15}$$

$$minimize: \quad \sum_{i=1}^{n} l_i \cdot (E_{S,i} + E_{T,i}). \tag{6.16}$$

[0137] Eqn. 6.14 is the constraint on execution time for SW; Eqn. 6.15 makes sure that it is done before the deadline meets; Eqn. 6.16 is the optimizing objective function. By solving that ILP problem, we can get an energy optimal, deadline satisfying task schedule.

Experimental results.

[0138] In total five cases are considered in our experiment.

• Case 1.

[0139] Conditions:

1. TB=1 symbol period;

2. $V_{LOW} = 1\,V, V_{HIGH} = 3\,V$;

3. The two timers and SW threads are scheduled independently.

[0140] The Eq. $C_{S,i}$ in Tab. 5 is the equivalent execution time on the 10MHz processor.

[0141] The optimal schedule result derived from the ILP solver is shown in Tab. 6. It is interesting to notice that though task 4 and task 12 have the same execution time, the scheduling result and energy consumed are quite different because of their different deadlines. For a stricter deadline, more part of the thread frame will be executed on the higher voltage processor and that means more power. Task 7 and task 14 also have the same execution time but different scheduling.

• Case 2.

43

[0142]   Conditions:

1.   TB=1 symbol period;

2. $V_{LOW} = 1V, V_{HIGH} = 3V$;

3.   The two timers are grouped together.

[0143]   The difference between case 1 and case 2 can be seen in Fig. 36, where the shadowed areas represent the processor time occupied by the DSTU thread nodes. In case 2, the two timers are integrated in one group and scheduled as a unit. Though there is idle time in that group, it is not available to the processes outside that group. The optimal schedule result derived from the ILP solver is shown in Tab. 8.

- Case 3.

[0144]   Conditions:

1.   TB=2 symbol periods;

2. $V_{LOW} = 1V, V_{HIGH} = 3V$;

3.   The two timers are grouped together for two symbols.

[0145]   All possible schedules of the timers in two symbol periods that make a reasonable distinction can be found in Fig. 37. The optimal schedule result derived from the ILP solver is shown in Tab. 10.

- Case 4.

[0146]   Conditions:

1.   TB=3 symbol periods;

2. $V_{LOW} = 1V, V_{HIGH} = 4V$;

3.   The two timers are grouped together for three symbols.

[0147]   The reason we change the higher voltage to 4V is that it is unschedulable with a 3V processor. The possible schedules of the timers for one TB is similar to case 3. The only difference is that one symbol, in which both timers are scheduled to L, is inserted between the original two symbols. The optimal schedule result derived from the ILP solver is shown in Tab. 12.

- Case 5.

[0148]   Conditions:

1.   TB=4 symbol periods;

44

2. $V_{LOW} = 1\,V, V_{HIGH} = 5\,V$;

3.   The two timers are grouped together for four symbols.

[0149]   For the same reason as in case 4, we change the higher voltage to 5V. The possible schedules of the timers for one TB are similar to case 3 except for the two inserted symbols. The schedule result derived from the ILP solver is shown in Tab. 14.

Further Analysis.

[0150]   Checking the result above carefully one will find that the energy cost of task7 and task11 in case2 is higher than those in case3. That is because we have more scheduling options in one TB for case3, which can be interpreted as a less power costly scheduling result but with more effort in overhead and communication.

[0151]   To get an idea of the energy cost versus different scheduling cases we can refer to Tab. 15. Remember case4 and case5 have a working voltage different from the first three cases to keep it still schedulable. To make a fair comparison, we change the high voltage of all cases to 5 V. The result is shown in Tab. 16. It can be found out that while the granularity of a TB increases, the energy consumption increases also. However, as mentioned before, the whole overhead cost will decrease since there are fewer TBs to be scheduled dynamically. We also computed the energy consumption under only one processor architecture. It's 18.3 when that processor works at 3 V and 51.2 when it works at 5 V. Compared with the one processor architecture, there is an energy saving of about 20% even at the biggest TB in case 5. This energy saving percentage varies with the system load. The heavier the load, the less the saving percentage because more work will be done on the high voltage processor. When all the tasks are done on the high voltage processor, it is reduced to the one processor case.

[0152]   The overhead of the above dynamical scheduling process can be divided into two parts. First, the computation to find an optimal scheduling at the coming of a new TF; secondly, the effort to control the thread nodes when a TF is being executed. An MILP model is chosen here for the first part and the branch and bound approach is used to solve that problem.

Conclusions.

[0153]   In this proposal, we present our approach of doing task concurrency management on a multiprocessor architecture for power saving consideration. The TCM is done in three steps, namely the concurrency extraction, the static scheduling and the dynamic scheduling steps. A static scheduling method is proposed to get an power optimal static scheduling for a thread frame under given time constraints or deadline. Varying the frame deadline will give a series of cycle budget and power cost tradeoff points, on which a dynamic thread frame scheduling will be done. At present, we used a MILP method for the dynamic scheduling. Other scheduling methods can be applied as well. This three step approach can bring flexibility and reduced design time to the embedded SW design.

[0154]   These techniques are applied to a ADSL modem application and some interesting results are derived. First, more than 20% power saving can be obtained. Second, we changed the granularity of a TF and demonstrate that a tradeoff exists between the dynamical scheduling overhead and scheduling optimality.

A new static scheduling heuristic and its experiment results on the IM1 player

[0155]   In this section we will introduce a new static scheduling heuristic. Its major difference from existing algorithms are explained then. Finally, it is applied to the IM1 player. Experiment results are used to illustrate the above arguments. The example is again derived from the IM1-player. The part scheduled here is similar to the one illustrated in the previous section (cfr. 6.3.1).

[0156]   Our approach can be applied to the multiprocessor platform without the need of changing the processor voltage dynamically. Given a multiple processor platform, i.e., the number of high-speed processors and low-speed processors, we aim at deriving an energy-cost *vs* time-budget curve. Compared with other scheduling algorithms, like MILP (mixed integer linear programming), our approach has the following difference. In the first place, our heuristic derives a set of working points on the energy-cost *vs* time-budget plane instead of only one point. These working points range from the most optimal performance point within that given platform to the point barely meeting timing constraints. For a given platform, these working points form a Pareto curve, which enables designer to trade off between cost(energy-consumption) and performance(time-budget). For example, when

combining subsystems into a complete system or during the dynamic scheduling stage, we need to select working points of subsystems and combine them into a globally optimal working point. Existence of these different working points is mostly due to the different *thread node* to processor assignments (some of it is due to idle time in the schedule). As a result this heuristic deal with both a crucial assignment problem and ordering problem. Secondly, it uses an intelligent policy to prune the search space heavily. Hence, the computation complexity is reduced. Thirdly, it does not need to change the processor voltage dynamically.

[0157] To achieve the above objective, the heuritic uses the following two criteria to make the assignement and scheduling decisions. The first criterion is the *self-weight* of a *thread node*. It is defined as the execution time of the *thread node* on the low-speed processor. The larger the self-weight of a *thread node*, the higher its priority to be mapped on a high-speed processor.

[0158] The second criterion is the *load* of a *thread node*. If some *thread nodes* are depending on a *thread node*, the sum of the self-weights of all the dependent *thread nodes* is defined as the load of the *thread node*. It is worth noting that "dependent" means control dependence. The more load a *thread node* has, the earlier it should start to execute.

[0159] From the above criteria, when a processor is available, the following strategy takes care of selecting a candidate *thread node*.

[0160] 1. When a *thread node* is dominant both in the self-weight and load over the other candidate *thread nodes*, it will be scheduled first.

[0161] 2. When one *thread node* has a dominant self-weight and another *thread node* has a dominant load, either of them can be scheduled first. By alternating their order different points on the energy-cost *vs* time-budget plane can be generated.

[0162] It is better to realize that the heuristic implicitly includes energy considerations. Because for a given processor, the energy consumption is directly related to the execution time. The *self-weight* and *load* in the heuristic are merely two interpretations of execution time from different perspectives. Applying this heuristic to the IM1 player will be discussed in the following section. Even though it seems relatively simple, it turns out to be very effective for scheduling the tasks in the MPEG-4 IM1 player.

[0163]   When applied to the IM1 player, the heuristic the task graph of the IM1 player as input. Two parellel ARM7TDMI processors, running at 1V and 3.3V respectively, are used to illustrate the scheduling. The main constraint for scheduling is the *period*. It starts at the reading of one VOP[2] data and expires at the end of the decoding of the data. Energy is the cost function used here. Currently, it includes both the processor energy and DTS energy. The DTS energy is mainly used for buffering between the reading and decoding of the data.

[0164]   The period we have measured for the initial V1 version is around 1000μs per packet. Such a long period is caused by reserving and releasing semaphores. The period varies due to the jitter of the operating system. After our initial transformations, we have produced version V2 of the task graph.

[0165]   Without loss of generality, we assume that one VOP data consists of two packets of data. The current implementation dictates that the two packets be read and decoded sequentially. That is, only after one packet is read and decoded, a second packet can be read and decoded. Due to such a limitation, we can only reduce the period by executing some *nodes* on a high VDD processor.

[0166]   For one *node*, we use t_L and t_H to denote the its execution time on the 1V ARM7TDMI and 3.3V ARM7TDMI respectively and Δt to denote the difference between these two execution times. Similarly, we use E_L and E_H to denote its cost of the processor energy to run on 1V ARM7TDMI and 3.3 ARM7TDMI and ΔE to denote the energy difference. Our calculations are based on the following formulae.

$$\Delta t = t_L - t_H = (1 - \frac{1}{3.3})t_L \qquad (6.17)$$

$$\Delta E = E_H - E_L = (3.3^2 - 1)E_L \qquad (6.18)$$

[0167]   For transformed graph V2, we start scheduling by assigning every *node* to the low Vdd processor and moving selected *nodes* to high Vdd processor when a shorter period is needed. We end up at assigning every *node* to the high Vdd processor. The period and total energy cost of the Pareto-optimal points in the search space are shown in Fig. 34.

---

[2] Video Object Plane

[0168] After additional transformations that lead to V3, we remove the implicit sequential restriction in the code. Consequently, it is feasible to read and decode more than one data packet concurrently. We can better utilize the two processors by exploiting the concurrency and hence we can shorten the period with less energy cost.

[0169] Two scheduling examples of the V3 task graph are shown in Fig. 35. We use $N_{1A}, N_{1B}\Lambda$ and N_{2A}, N_{2B}, Λ to denote the *nodes* for reading and decoding of the two data packets respectively. Fig. 35 shows that the period varies significantly due to different scheduling. The scheduling result is shown in Fig. 34.

[0170] Fig. 34 shows that scheduling point 2, 7 and 8 are not on the Pareto curve because point 3 and 9 offer a shorter period with a lower energy cost. It is clear that version V3 is better than version V2.

[0171] Notice that reading two data packets concurrently has a higher memory energy cost than reading the packets sequentially since a larger buffer is needed. In the graph, the total energy cost for version V3 is initially higher due to the above reason. At smaller period points, scheduling the transformed graph V3 offers a shorter period with a lower energy cost.

Management of Data Structures at the Task-level

[0172] The data used by the tasks needs to be managed and mapped efficiently on the target platform at the task-level. More in particular, we propose several transformations that optimize the energy consumption and memory occupation of data structures which dependent on the dynamic behavior of the application. Note that these transformations will be applied on a system using the grey-box model presented above. The main advantage of this model is that is exposes all relevant information about the dynamic behavior and the most important data structures of the application.

Reducing the Dynamic Memory Requirements

[0173] When developing an initial model of a system, specification engineers normally spend neither time nor energy to accurately use dynamic memory. They are mainly concerned with the functionality and the correctness of the system. As a result, the software model is often very memory inefficient. However, for the design engineers it afterwards

becomes an almost unmanageable and at least an error-prone process to improve the dynamic memory usage due to dangling pointers and memory leaks. Therefore, they will either redesign the system from scratch or use the same system but increase the memory, which results in a more power hungry architecture. We will present two formalizable transformations that allow us to systematically reduce the amount of dynamic memory required by the application. The benefit of these transformations can be exploited on different multi-processor platforms. Therefore, we will discuss the influence of the platform on the DM requirements of an application and we will explain how these can be relaxed.

[0174] **Control/Data-flow Transformations Enabling a Reduction of the Dynamic Memory Requirements.** With the first transformation we reduce the life-time of the dynamically allocated memory by allocating the memory as late as possible and deallocating it as soon as possible. As a result of our approach, we are able to increase the storage rotation, which can significantly reduce the amount of memory required by the application. Indeed, in most applications memory is allocated in advance and only used a long time after. This design principle is applied to obtain flexible and well structured code which is easy to reuse. Although this can be a good strategy to specify systems on a powerful general purpose platform, the extra memory cost paid is not acceptable for cost-sensitive embedded systems. This allocation policy could also be implemented using a garbage collector. However, transforming the source code instead of relying on a run-time garbage collector to implement this results in a lower power consumption. The transformation acts as an enabling step for the second transformation and the task-ordering step discussed below.

[0175] By postponing the allocation we are able to gather more accurate information on the precise amount of memory needed. This information can be used to optimize the amount of memory allocated for each data type. The ratio behind this transformation is that specification engineers often allocate conservatively large amounts of memory. In this way, they try to avoid cumbersome and difficult debugging. They hereby forget that it is much harder for implementation engineers to re-specify afterwards the amount of memory as they do not have the same algorithmic background. For the same reason, automation of this transformation is particularly hard.

[0176] We have applied both transformations on the dynamically allocated buffers in the IM1-player. These buffers are used to store the input and output of the compression layer. The size of the input buffers is transmitted in the OD data packets; the size of the output buffers is fixed at design-time and differs for each decoder type.

[0177] We have profiled the time-instants at which each buffer is (de)allocated and is first(last) used. All the input buffers are allocated and freed at the same time. However, on the original one processor-platform, the buffers are used sequentially. As a consequence, by reducing the life-time of these buffers it is possible to overlap them and significantly decrease the amount of memory needed. This principal is illustrated on the left part of Fig. 38.

[0178] By analyzing the memory accesses to the allocated memory, we have found out that a large part of the dynamically allocated output buffers is never accessed. By applying our first transformation, we are able to reduce the output buffer size by making it data dependent. Instead of using worst case sizes, the more average data dependent sizes required by the input stream can now be used (see right part of Fig. 38).

[0179] **Constraints on Static and Dynamic Task-Schedule Reducing the Dynamic Memory Cost on a Multi-Processor Platform.** After improving the characteristics of the application with the transformations mentioned above, its tasks still need to be assigned and scheduled on a 'platform.' By constraining the task-schedule of the application (e.g. scheduling the task that requires the largest amount of memory first) it is possible to avoid consecutive (de)allocations of the same memory. In this way, not only some memory accesses can be avoided (and as a result power saved), but also the fragmentation of the memory can be reduced.

[0180] In addition, when exploiting the parallelism inside the application, the amount of memory needed by the application will normally increase proportional to the amount of processors. This relation can be improved by avoiding that two tasks both using a large amount of memory, are executed simultaneously. Our transformations applied prior to this step will relax the constraints on the task-schedule.

[0181] We will illustrate this with the same example as above. Initially, it is useless to reorder the different tasks to decrease the DM cost of the input-buffers. All buffers

51

have the same life-time and no gain can be made by reordering the different tasks. After applying the enabling transformations discussed above, we obtain more freedom to reorder(or constrain) the task-schedule to decrease the DM cost. E.g. we can avoid decoding simultaneously the wavelet decoders that require much memory. As a consequence, the total amount of required memory increases less than proportional to the amount of parallelism (see Fig. 39).

Experiments and Results

[0182]   In this section we will prove quantatively the results of our transformation methodology. In a first part we will present the benefit of our transformations on the input buffers. In a second part we will briefly explain the potential benefit of our methodology on the output buffers.

[0183]   We will schedule the IM1-player on a representative platform consisting of two software processors with extra hardware accelerators added to decode the AV-data. The system layer of the player consists of three tasks running on the software processors, i.e. the delivery layer, BIFS- and OD decoder. The two parallel SA110 processors running both at 1V are used to illustrate the scheduling. The main constraint for scheduling is the *time-budget*, i.e. 30ms to render one frame. The two SA110 processors consume 40mW each. The energy consumed by the hardware accelerators is estimated based on realistic numbers that we obtained by measuring the a wavelet decoding oriented chip that we have developed at IMEC. The cost of the memory is calculated based on recent memory-models supplied by vendors (Alcatel 0.35u DRAM for the embedded memory and Siemens Infineon DRAM for the external memory).

[0184]   In a first experiment we have scheduled the IM1-player using only one hardware accelerator. We have added afterwards an extra one to improve the performance of the system and decode more objects per frame. We have compared all our results with the performance of the initial specification mapped on the same platform.

[0185]   The results in Tab. 18 and 17 clearly show the potential benefit of the transformations as they allow us to reduce the global power cost(including both processor and memory cost) with almost 20%. Moreover, this methodology still can be applied on other parts of the player and does not need to be restricted to the input buffers only. We have

obtained this power gain because we are able to significantly reduce the amount of memory required by the application. As a consequence, the data can be stored on embedded DRAM instead of on off-chip DRAM. We believe that two reasons make our approach scalable to future applications: if embedded DRAM would become available that is large enough to contain the initial data structures, our transformations will still be needed because the power consumption of embedded DRAM scales at least super logarithmic with its size. Moreover, we believe that the amount of data required by the applications will increase at least proportional to the size of embedded DRAM.

[0186] As indicated above, the buffer size would normally increase proportionally with the amount of parallelism. By constraining the schedule this can be avoided. The results for a two hardware accelerator context are represented in Tab. 18. One can also derive from Tab. 17 and Tab. 18 that the fraction of the memory cost increases in the total energy budget when more parallelism is exploited in the system. This illustrates the importance of these constraints on highly parallel platforms.

[0187] In this final paragraph we present the effect of our transformations on the output buffers. We are able to significantly reduce the size of these buffers, i.e. from 750kB to 105kB. Consequentely, it further reduces the power cost of the memory accesses and allows us to embed the IM1 protocol on a portable device.

Conclusion

[0188] In this paper we have presented several systematic data/control flow transformations for advanced real-time multi-media applications which reduce the power (area) cost of the dynamically allocated memory or increase the performance of the system. The first transformation has reduced the life-time of the dynamically allocated data structures. As a result, the memory rotation could be significantly increased which reduces the required amount of memory. It also functioned as as an enabling step for the second transformation, in which we have defined more accurately the amount of memory needed for each data structure. Finally, we have given some constraints which help to relax the effect of parallel execution of tasks on the increasing buffer cost. We have illustrated these techniques with examples extracted from a real multi-media application, the MPEG4 IM1-player. They allowed us to reduce the energy consumption of the memories with at least a factor of 5. As a

53

consequence, the global power cost of a significant part of this particular application has decreased with 20% without even touching the other subsystems.

Conclusion

[0189]   It is clear from the description and Fig. 1 that the invented design flow differs from the prior art flows as it include an extra task concurrency optimization step being located before hardware/software co-design. Further the new design flow works on a grey-box model, containing only part of the logical operations and the co-design step starts of with a partial scheduled description. The TCM step also generates a real-time operating system and a dynamic scheduler therefor Fig. 2 illustrates that the TCM optimization step comprises of a design-time and run-time step. Also extra step, such as grey-box description extraction, dynamic memory management, task-level data transfer and storage, and task concurrency improvement steps are indicated. It is shown that the static or design-time step generates static schedules and pareto information, being used within the run-time scheduler. Fig. 3 shows a possible implementation of the real-time operating system with a dynamic scheduler,

accepting a request for a new schedule. A new task is loaded. For each of tasks on the list on alive tasks a subset of all possible intra-task schedules, more in particular the Pareto optimal ones, are loaded. The dynamic scheduler selects for each of these alive task one schedule. The selected schedules define an overall schedule for the alive tasks. This schedule is executed, starting with executing the first scheduled node of the first scheduled task.

Table 1

	Execution Time							Energy Consumption						
	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_0	t_1	t_2	t_3	t_4	t_5	t_6
P1(H)	3	10	12	13	16	13	30	27	90	108	117	144	117	270
P2(L)	9	30	36	39	48	39	90	3	10	12	13	16	13	30

Table 1

Table 2

	Thread Frame One			Thread Frame Two		
	opt.1	opt.2	opt.3	opt.1	opt.2	opt.3
Cycle Budget	20	60	100	40	60	80
Energy Cost	110	80	50	90	60	50

Table 2

Table 3

Vdd	1 V	3 V	4 V	5 V
Frequency	10MHZ	30MHZ	40MHZ	50MHZ
Power (normalized)	1	27	64	125

Table 3

Table 4

Task	4	5	6	7	8	9	10	11	12	14	15	16	17	18
Deadline (symbol)	128	96	128	128	768	256	128	132	16	64	16	2048	64	576
Ex. Time (ms)(10MHZ)	3	3	3	21	240	9	60	12	3	21	3	864	18	285

Table 4

Table 5

(L,H)	1,1		2,_		_,2	
	s,_	_,s	s,_	_,s	s,_	_,s
Eq. $C_{S,i}(\mu s)$	166	624	102	690	230	564
$E_{S,i}+E_{T,i}(\times 10^{-6})$	797	6247	230	6338	1364	6210

Table 6

(L,H)	1,1	2,_		_,2		Energy $(\times 10^{-6})$
	s,_	s,_	_,s	s,_	_,s	
task4	1					6900
task7	1	65	62			100315
task8	1		138	629		1.73×10^{6}
task9		89				20470
task10		2	67	59		505582
task11		3	11	18		94960
task12		5	11			16154
task14		1	14	49		155798
task16	1	2	855	1190		7.04×10^{6}
task17		3	8	53		123686
task18	1	1	332	242		2.44×10^{6}
					Total	1.22×10^{7}

Table 7

(L,H)	1,1		2,_		_,2	
	s,_	_,s	s,_	_,s	s,_	_,s
Eq. $C_{S,i}(\mu s)$	166	498	102	306	188	564
$E_{S,i}+E_{T,i}(\times 10^{-6})$	797	5113	230	2882	1322	6210

Table 8

(L,H)	1,1		2,_		_,2		Energy (×10⁻⁶)
	s,_	_,s	s,_	_,s	s,_	_,s	
task4			30				6900
task7	123		4		1		100273
task8	429	339					2.075×10^6
task9			89				20470
task10	10	117	1				606421
task11	12	19				1	112921
task12	14	1			1		17593
task14	32	30			1	1	186426
task16	467	1579		2			8.45×10^6
task17	40	22		1	1		147478
task18	4	570		1	1		2.92×10^6
						Total	1.464×10^7

Table 8

(L,H)	22,__		21,_1		11,11		1,_12		_,22	
	s,_	_,s	s,_	_,s	s,_	_,s	s,_	_,s	s,_	_,s
Eq. $C_{S,i}$ (µs)	204	612	268	804	332	996	352	1056	376	1128
$E_{S,i} + E_{T,i}$ (×10⁻⁶)	460	5764	1027	7975	1594	10226	2117	11269	2644	12420

Table 9

Table 10

(L,H)	22,—		21,—,1		11,11		1,—,12		—,22		Energy $(\times10^{-6})$
	s,—	—,s	s,—	—,s	s,—	—,s	s,—	—,s	s,—	—,s	
task4	15										6900
task7			4		69		1				100271
task8				2	213	168	1				2.076×10^6
task9	45										20700
task10				2	5	57					606802
task11					6	9		1			112867
task12			2		5						17999
task14			1		14	15	1				186825
task16					234	787	1	2			8.446×10^6
task17					20	10	1	1			147526
task18					2	284	1			1	2.92×10^6
										Total	1.464×10^7

Table 10

(L,H)	222,—		221,—,1		121,1,—1		12,—,1,2		—,2,2,2	
	s,—	—,s	s,—	—,s	s,—	—,s	s,—	—,s	s,—	—,s
Eq. $C_{s,i}(\mu s)$	204	816	268	1072	332	1328	364	1456	396	1584
$E_{s,i}+E_{T,i}(\times10^{-6})$	588	13440	1612	18496	2636	23552	3628	26560	4620	29568

Table 11

59

Table 12

(L,H)	222,_		221,_1		121,1_1		12_,1_2		_2_,2_2		Energy $(\times10^{-6})$
	s,_	_,s	s,_	_,s	s,_	_,s	s,_	_,s	s,_	_,s	
task4	15										8820
task7					33	7	1			1	260100
task8		1			99	154	1	1			3.93×10^6
task9	45										26460
task10						25		1		16	1.09×10^6
task11				1	1	8					209548
task12		1			2	1					42852
task14					5	14	1		1		351156
task16					41	639	1	1			1.52×10^7
task17					8	11	1		1		288408
task18						74		1	117		5.23×10^6
										Total	2.665×10^7

Table 13

(L,H)	2222,_		2221,_1		1221,1_1		122,1_2		22,2_2_2	
	s,_	_,s	s,_	_,s	s,_	_,s	s,_	_,s	s,_	_,s
Eq. $C_{S,i}(\mu s)$	204	1020	268	1340	332	1660	370	1850	408	2040
$E_{S,i}+E_{T,i}(\times10^{-6})$	716	26012	2065	35297	3710	44878	5309	51189	6908	57500

(L,H)	2222,__		2221,__		1221,1__		122,1__2		22,_,2__2		Energy (×10⁻⁶)
	s,_	_,s	s,_	_,s	s,_	_,s	s,_	_,s	s,_	_,s	
task4	15										10740
task7				1	33	7			1		441681
task8			1	1	57	132	1				6.18×10⁶
task9	45										32220
task10				1				24		7	1.67×10⁶
task11				4		4					320700
task12			1		1		1			1	68584
task14				1	3	11			1		546993
task16				1		437		72		2	2.34×10⁷
task17				6	5	5					454722
task18								46		98	7.99×10⁶
										Total	4.11×10⁷

Table 14

Case Number	1	2	3	4	5
Total Energy Cost	12.2	14.64	14.64	26.65	41.1

Table 15

Case Number	1	2	3	4	5
Total Energy Cost	31.2	34.6	34.6	39.1	41.1

Table 16

	1HW accelerator		2HW accelerators	
	Execution Time	Processor Energy	Execution Time	Processor Energy
OD	9ms	0.37mJ	9ms	0.37mJ
BIFS	24ms	0.98mJ	24ms	0.98mJ
Delivery	8.1ms	0.32mJ	16.2ms	0.64mJ
Wavelet	30ms	1mJ	30ms	2mJ
Total	30ms	2.58mJ	30ms	4mJ

Table 17

	1HW accelerator			2HW accelerators		
	Mem. Accesses	Mem. Size Pre	Mem. Size Post	Mem. Accesses	Mem. Size Pre	Mem. Sixe Post
OD	0.58k	10kB	0.58kB	0.58kB	10kB	0.58kB
BIFS	2.41k	41kB	2.41kB	2.41k	41kB	2.41kB
Delivery	35.9k	35.9kB	12.4kB	71.8k	71.8kB	17kB
Wavelet	35.9k	35.9kB	12.4kB	71.8k	71.8kB	17kB
Total	74.9k	86.9kB	14.8k	146k	193kB	19.4k

1HW accelerator		2HW accelerators	
Energy Pre	Energy Post	Energy Pre	Energy Post
0.78mJ	0.16mJ	1.54mJ	0.19mJ

Table 18